컴포넌트 워크플로우 커스터마이제이션 기법
(A Component Workflow Customization Technique)

김 철 진 † 김 수 동 ††
(Chul Jin Kim) (Soo Dong Kim)

요 약 소프트웨어를 개발하는데 미리 구현된 블록을 사용하여 소프트웨어 개발 비용과 시간을 단축할 수 있다. 이와 같이 미리 구현된 블록을 컴포넌트(Component)라고 하며 컴포넌트는 실행 단위로 개발자에게 인터페이스만을 제공하여 내부 상세한 부분을 숨기므로 쉽고 빠르게 대형 어플리케이션을 개발할 수 있다. 개발자는 완전히 내부를 볼 수 없는 블랙 박스(Black Box) 형태의 컴포넌트를 사용한다. 그러나 개발자들은 개발 도메인의 특성에 맞게 속성 및 워크플로우(Workflow)의 변경을 원하기 때문에 커스터마이즈(Customize)할 수 있는 방법이 있어야 한다. 기존의 커스터마이즈 기법은 컴포넌트의 속성을 변경하는 것에 국한 되어 있다. 본 논문에서는 비즈니스 측면에서 속성 뿐만이 아니라 컴포넌트 내부의 워크플로우도 변경할 수 있는 기법을 제시한다. 기존에 워크플로우를 변경한다는 것은 컴포넌트 내부를 개발자가 이해하고 코드 수준에서 수정해야 하는 화이트 박스(White Box)이지만, 본 논문에서는 워크플로우의 변경을 화이트 박스가 아니라 블랙 박스 형태로 컴포넌트 인터페이스 만을 이용해 커스터마이즈 할 수 있다. 본 논문에서 제시하는 컴포넌트 커스터마이즈 기법은 특정 비즈니스 측면에서 도메인에 종속적인 특성을 가지며 컴포넌트를 좀더 범용적으로 사용할 수 있는 향상된 커스터마이즈 기법을 제시한다.

Abstract When the developers develop the software, the cost and time of the software development can be reduced by using blocks that are implemented previously. We call these implemented blocks components. They provide only the interfaces of component to developers, while the detailed internal information is being hidden. The components can be used to develop large applications fast and easily. Developers use the black-boxed components that the internal details of components is invisible. But, Developers want the properties and workflows of components to be modified. To do this, the customization technique is needed. The existing customization techniques are limited in the customization of component properties. In this paper, it proposes the technique that can change the workflow of component as well as properties in the business point of view. In the existing workflow customization technique which is called white box component, the internal details of component should be understood by developers and should be modified in code level. In this paper, we propose a customization technique which using the component interface in black box form only. This customization technique can be used more generally in a business field.

1. Introduction

In the mid- to late-1980s, component-like systems began to appear , but they were not called components, as such. Arguably Xerox's Parc's early document mixing, Mach's operating system modules, Ada's class modules, and Applet's QuickTime were all steps toward a component model[1]. Advancing these reusable components is the reason why the object-oriented development hasn't improved the cost of development and the quality of the software product. That is to say, the object did not provide the basic reuse, but the components, such as the OLE of Microsoft, become important in software market quickly for its reusability.

† 비 회 원 : 숭실대학교 컴퓨터학과
cjkim@selab.soongsil.ac.kr
†† 종신회원 : 숭실대학교 컴퓨터학과 교수
sdkim@computing.soongsil.ac.kr
논문접수 : 1999년 8월 23일
심사완료 : 2000년 3월 24일

The components mean the industrialization of software development and the possibility of mass production. Although the components have generality, they need mechanisms to adapt to the feature domains when reuse is in the special domain. The existing customization only changes the component's properties that are focused on the visual component mostly.

To reuse the component in the business logic as well as the graphic user interface, the change of property and the component internal workflow is become possible. The component internal workflow should be changed using only interface. In this paper we propose a customization technique to change the properties and workflow of component using only component interface without altering the component interior.

2. Related Work

2.1 Component and Component Model

Until now, though we have heard about components in fourth generation language such as Visual Basic and Delphi, these components have many limitations to give definition to the black box components. We give a definition of component using several common definitions.

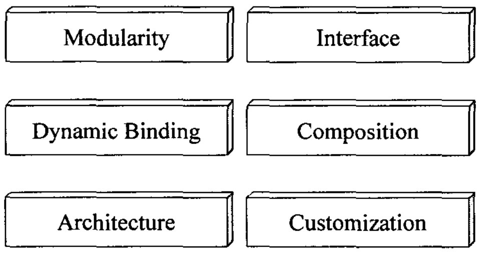


Fig. 1 The common feature of component

The common features of components can sum up modularity, interface, dynamic binding, composition, architecture, customization and so on, like Figure 1.

Having a strong modularity is equal to have a strong generality, the general components mean that software can be developed in plug and play like hardware developed in various domain without changing the properties of component much.

When the existing library is used by the application, the dynamic binding calls is only uni-direction, that is to say, from application to library. However the calls in component is called bi-direction calls by dynamic binding. The components can be connected to the application in runtime using interface and can also be connected with other component dynamically. The library must be recompiled in order to reuse in the development system, but the components can be used dynamically without recompiling in runtime.

The components in the reusable unit of the black-box need not to recognize the internal mechanism, but it must know the interface of component in order to use component in the development system. The developers use the interface in among components as well as in the development systems. The interfaces are defined as the services of component, that is, the interfaces are implemented logically by the internal classes of component.

The components must be designed and implemented to meet the previously defined architecture so that it can interact with other components or framework. The component-based architecture is offered in the form of framework to archive the functional improvement adding or replacing components. The component architectures like CORBA or DCOM is substituted for the operations of components. The components should be designed to meet the component architecture and should be substituted for the facilities ,such as transaction, state management, security management, and so on. The components have the architecture isn't able to operate on other component architecture. The components depend on the features of architecture, but the component model like CORBA and DCOM has released the specifications for interoperability among components[2].

The components can plug in and compose component though interfaces. These compositions need designing the action among components and the components must be composed in runtime. At present in software market there are Visual Java(SUN), Visual Age(IBM), and so on, as tools for component composition, but they don't support creating, deleting, and connecting dynamically and don't check the validation in semantic level[3].

When the developer develops the application using the components, he can set the components in a developing system to change the properties of component. These customizations generally access the component internal datum using interfaces[4].

The component model defines the basic architecture, the interfaces of component and the mechanisms for the interoperation between components and container. Like this, the component model defines the environment for supporting reusable component[5].

The components operate within a container that support the application context for more than one component. It also supports services to manage and control the components. Commonly the client components operate within the visual container and the server components operate within non-visual container that supports the services, such as TP(Transaction Processing) monitor, Web server and database system[6][7].

The JavaBeans component model supports the development of the reusable Java components and is designed to run through the Java development tools. The developers who use the Java development tools can customize the JavaBeans using the property tables or the customization methods.

2.2 The Trade-Off of the Component Customization

We can consider two things about components, which is granularity and generality of the components. If the size of the components increases in volume, the applicable areas are narrowed. So the generality goes down. Like this, the trade-off among the components is in existence and the developers can select to meet the scope of component by them.

A Figure 2-(A) is shown the relationships among the generality, the granularity, and the customization of components. The larger the size of component is, the more the alteration it has. So if the number of the customization is much more, the generality of the component will be decreased. In the industrial world the developers want larger components than smaller ones. However the large components decrease the generality. To increase the generality must support the customization technique that can be used the components generally by the application developers.

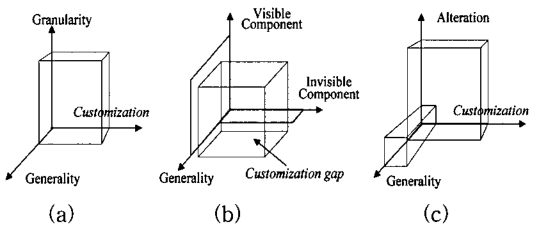


Fig. 2 The Trade-off of the Customization

The generality of the visible components is stronger than the invisible components. The invisible components are mostly the related with the business logic that has many change points. So it's generality is weak. As Figure 2-(B), the invisible components are much more customized than the visible components. The visible components customize the states of component mostly, but the invisible components customize the internal facility of component

As Figure 2-(C), the customization has the constant proportion relationships with the alteration of components. If the customizable functionality is much more, the generality will be weak. On the contrary, if the customizable functionality is a few, the alteration will be weak and the generality will be raised.

2.3 The Property Customization

The developers can plug-in the components properly in the developing system to change the properties of component when the application is developed using the components. These customizations generally use the interfaces to access the internal datum of the component. The properties of component changes in the design time or the run time and the changeable properties are defined by the getter or setter methods[2]. The developers who develop builder tools can provide the customization properties to the users so that they can get to know the interfaces of component easily by the design pattern of the getter or setter method.

```
public <STATETYPE> get<STATENAME>( );
public void set<STATENAME>(<STATETYPE>
data);
```

Like the code above change the type of property into return type for the getter method of property and append a property name followed by get to make a property. The getter methods don't need the parameters and the setter methods get the property values. The setter methods establish the property values of component via only parameters without return values. The parameter type is the property type and the name of method is consisted of set followed by property name.

```
public class Button {
    private String UIClassID = "ButtonUI";
    private Color color = Color.gray;

public String getUIClassID()
    {     return "ButtonUI";     }

public void setUIClassID(String UIClassID)
    {     this.UIClassID = UIClassID;     }

public Color getColor()
    {     return color;     }

public void setColor(Color  color)
    {     this.color = color;     }
}
```

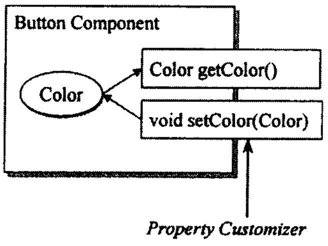


Fig. 3 Button Component

The above code defines getter/setter method that is used to customize the properties in the Button component. The color of Button has the default value that is Color.gray. If the color of Button should change into the blue, we can customize the property using the getter method.

The Figure 3 shows the relationship between the getter and the setter method of the Button component. The setter method is the interface of property input and the getter method is the interface of property output. Like this, the getter/setter pattern defines the property methods and it can be customized to make the properties easily to the application developers or to the builder tools using the components.

3. The Component Model

To apply the component customization technique which have been proposed in this paper the component model as the Figure 4. This component model is based on the component model of JavaBeans and consists of several component classes in the interior. Each class is developed by the component developers and is packaged via the manifest file which is specified the classes. To access the component's internal classes we use the interface of the component and consists of the property interfaces and the interfaces to customize the internal work flows of the component.

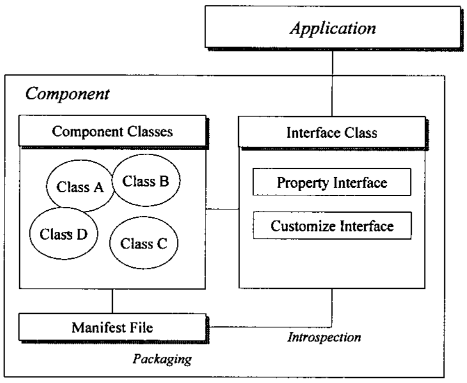


Fig. 4 The Component Model

When the component users develop the applications by using the component, they use the interfaces of component whose the representation can be indicated as Figure 4.

The interface technique of IDL(Interface Definition Language) form declare only the signature of method, such as IDL, without implementing the methods of

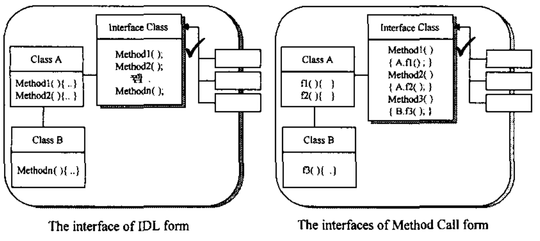


Fig. 5 The Component Interfacing Technique

the interface class which implement in the internal classes of component[8]. The method call form is technique that calls directly the methods of the internal class of component in the interface class. At this time the interface class is not the class that is declared in the IDL form but the concrete class that can be instantiated the objects. The component interface technique using in this paper is the second way and the concrete interface class is contained the interior of component.

The interface that affected to the change of workflow among the component classes adds two arguments.

public function(varType$_{1-n}$ variable$_{1-n}$, String ClassName, boolean flag);

Like above, the function appends the ClassName and the flag in the arguments. The ClassName is the class name that has the desired flow by developers. The flag is argument to determine whether the workflow is customized or the default workflow used. The interfaces of component include the general interface and the special interface that changes the workflow of the internal classes of component.

The package technique uses the JAR(Java Archive) file form of Java. The JAR file includes the class files for Java applet and application and the related file such as image file, audio file, text file, and so on[ ]. The manifest file is used as packaging the component classes in the JAR file. The manifest file specifies the class name behind Name: like Figure 6.

The documentation uses the manifest file to support API. The interface of the component uses the manifest file to extract the classes of the component in the runtime[9].

```
Manifest-Version: 1.0

Name: Client.class
Name: Server.class
Name: DBManager.class
Name: FileManager.class
Name: LoginComponentInterface.class
```

Fig. 6 The Manifest File

4. The Workflow Customization Technique

If the visual component has the property customization facilities that change only the property of component, the components can be reused in application independently. However the non-visual components are restricted to reuse with only the property customization facilities in the business point of view.

As the component includes many classes, it exists the workflows among the classes. These workflows mean that the component contains the flow of function. If the component is provided the function to customize the internal flow of component as well as the component properties, it will be reused in the various domain generally. The component is the reusable unit of the black-boxed that can be reused with only the interfaces of component without recognizing the interior of component[10][11][12].

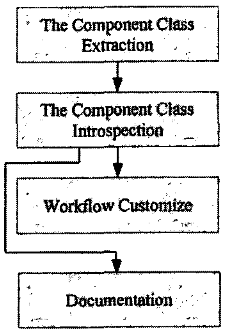

The Component Class Extraction
The Component Class Introspection
Workflow Customize
Documentation

Fig. 7 The Workflow Customization Process

Like Figure 7, the workflow customization of component can extract the packaged classes of component using the Reflection in the runtime. The

Reflection facility can get the introspection information of the extracted class. The extracted introspection information is class name, constructor, attribute, function name, and so on. The component interfaces among the extracted classes can change the workflow of component.

4.1 The Component Classes Extraction

The component can consist into only a class. If the component consist into several classes, it includes the manifest file that represents the class information of component. The manifest file can represent whether what classes within the component exist.

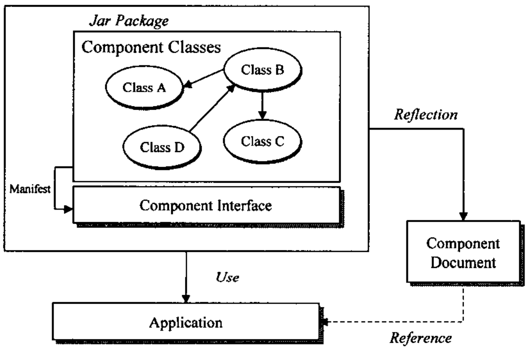


Fig. 8 The Class Extraction of Component

Like Figure 8 the packaged component include the interface class that represents the class information within the component and the application developer uses the interface class. The component interface refers to the manifest file to document the component and support it to the application developers. After all, the component interface recognizes the internal classes of component to customize the component. The workflows of component can implement the default workflow within the interior of component when the component is developed, but they can be customized via the component interface when the application is developed

4.2 The Extraction of Class Information

To get the information of the extracted classes from component in the runtime uses the Reflection facility of Java. The Reflection facility appropriates in the case of executing the internal class's method of component and supports the proper methods to customize the workflow of component.

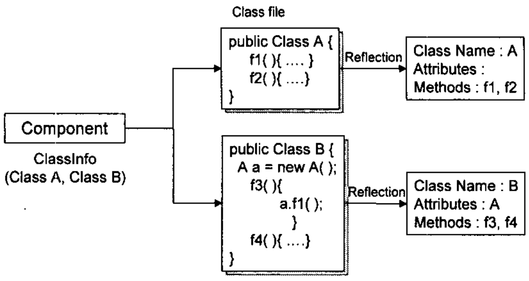


Fig. 9 The Extraction of the Class Information

The extracted information through the reflection facility is the information that needs to customize the existing workflow among the internal classes of component. As Figure 9, The class A and B are the internal class of the component that is extracted from the component. They show their information in the runtime using the reflection facility of the Java.

Class Name	State	Method

A		f1(·, Object, flag)
		f2(·, Object, flag)

Class Name	State	Method

B	A	f3(·, Object, flag)
		f4(·, Object, flag)

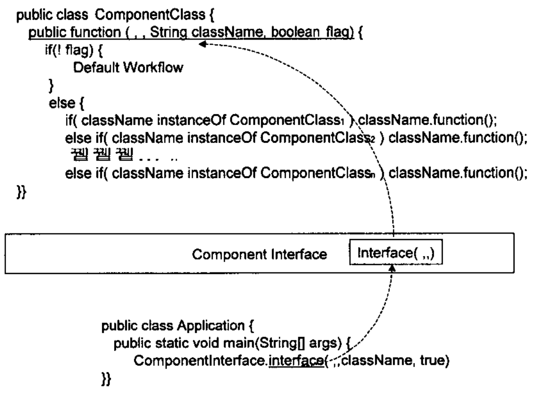
Fig. 10 The Reflection Information of Class A and B

Like Figure 10, the methods of the internal class have two arguments that appended when the component created. The argument Object is object to customize the workflow among the classes of the component and the argument flag is the flag to determine whether the workflow is customized (True) or the default workflow is used (False).

4.3 The Workflow Customization

The component user can customize the workflow of the component only using the interface of component. In this paper proposes three way to customize the workflow.

The first is the customization technique that chooses the desired workflow among the permissible workflows through the interface of component. If the component user want to customize the workflow of component, he should specify the desired class name to the argument of the component interface. The component user can refer what the permissible class exists through the component document.



```
public class ComponentClass {
  public function ( , , String className, boolean flag) {
    if(! flag) {
        Default Workflow
    }
    else {
      if( className instanceOf ComponentClass1 ) className.function();
      else if( className instanceOf ComponentClass2 ) className.function();
       궬 궬 궬 . . .  ..
      else if( className instanceOf ComponentClassn ) className.function();
}}
```

```
public class Application {
  public static void main(String[] args) {
    ComponentInterface.interface( ,,className, true)
}}
```

Fig. 11 The Selection of the Permissible Workflow

The Figure 11 shows the process that customizes the workflow of component through the component interface. If the application calls only the interface of component, the component customizes the workflow of classes automatically within the interior of component. If the developer does not want to customize the workflow of component, the flag is set to the false to use the default workflow. In the Figure 11, if the application calls the interface method of the ComponentInterface class, the interface of the component invokes the class among the internal classes. The method of the invoked class creates the object of the class name that is inserted in the argument. If the inserted class is the class of the ComponentClass$_2$, the object of class invokes the function() method. The name of method conforms the pattern that has the same method name among the classes

As Figure 12, the permissible class that can be insert into the component is the first class of

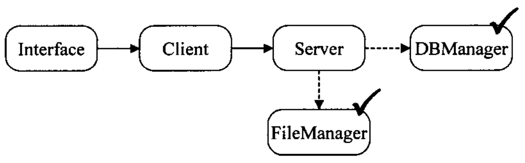


Fig. 12 The Permissible Workflow Classes

workflows which is FileManager or DBManager. The workflow of the DBManager is the flow that can access to the database and the workflow of the FileManager is the flow that can access to the file. These flows are determined at the Interface class that is the interface of the component. The interface has the appended arguments as Figure 11.

The second way is technique that changes the sequential flow of the permissible workflow. As Figure 13, the internal workflow of the component can be changed the flow of any workflow to the other workflow through interface.

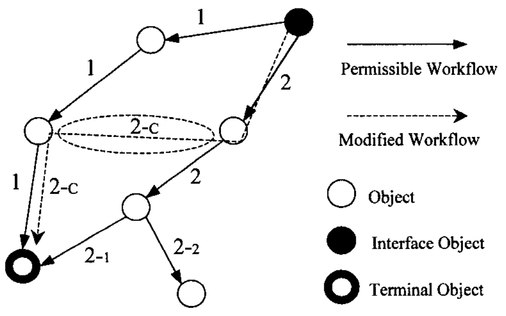
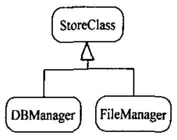


Fig. 13 The Customization of the Permissible Workflow

The workflow of Figure 13 consists of the 1, 2-$_1$ and 2-$_2$.' These workflows are the permissible workflows that are provided by the component. If the component user wants to change on the way of the 2 to the workflow of the 1, it create the workflow to 2-c dynamically. To connect the workflow between the 1 and the 2, the objects should pass the object to each other dynamically.

However these permissible workflow should open the interior of the component. So this technique supports not in black box but in white box to the developers.



```
abstract class StoreClass {
  퀠 .
  abstract void changeNode( );
  퀠 .
}
```

```
public class DBManager {
  public void changeNode( Object[] object ){
        Object[] obj;
        for( int i = 1; i < object.length() ; i++) {
              obj[i-1] = object[i];
        }
        searchData(id,pw);
        object[0].changeNode( obj )
  }
}
```

Fig. 14 Abstract class and subclass

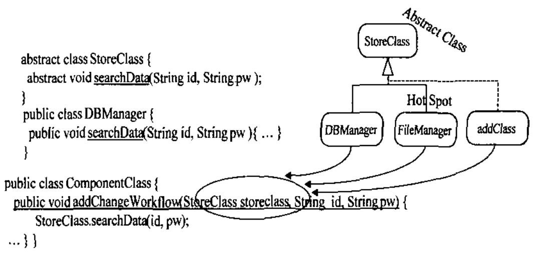
The Figure 14 is the abstract class StoreClass and the subclass DBManager. This code shows the way that changes the permissible workflow by the DBManager class. The changeNode(Object[] object) of the DBManager class receives the objects which want to change the workflow of the component. These objects are the flow of component. The sequence of workflow executes the process (searchData(id,pw)) of DBManager class and passes the objects which determine the next workflow to the next object(object[0]). These objects can change the flow among the permissible workflow.

```
public class ComponentClass {
  public function ( , , Object[ ] object,  boolean flag) {
      if( flag = "True") {
         Object[] obj;
         for( int i = 1; i < object.length() ; i++) {
               obj[i-1] = object[i];
         }
         object[0].changeNode( obj )      } } }
```

Fig. 15 ComponentClass

The Figure 15 shows use the component in the application. The method function() of the component class ComponentClass receives the objects which can change the workflow. The objects execute to themselves changeNode() which has the special function. After the object executes the function, it passes the array of object to the next object. The component use must recognize the structure among the objects and the permissible workflow of the component to insert the objects that represent the workflows.

The workflow customization technique of the third is the technique that changes the workflow inserting the object of the permissible workflow or the created object dynamically. The classes that have many the change of workflow extract to the abstract class and dynamically can change the workflow. The interface receives the object type of the abstract class that can change the workflow of component dynamically.



```
abstract class StoreClass {
  abstract void searchData(String id, String pw );
}
public class DBManager {
  public void searchData(String id, String pw ){ ... }
}
public class ComponentClass {
 public void addChangeWorkflow(StoreClass storeclass, String  id, String pw) {
    StoreClass.searchData(id, pw);
 ... } }
```

Fig. 16 The Dynamic Workflow Customization

Like Figure 16 the component makes the abstract class of the changeable part and the subclass must define the abstract method of the abstract class. The subclass has the common method among subclasses and the common method is specified with the regular pattern by the component developers. These subclasses can be accessed through the interface. The application developers insert the desired class to interface that can access the internal subclass of component. The interface of component declares the abstract class for the type of arguments to input the class of the workflow dynamically. The component can use the inserted object within the component by the substitution concept of the abstract class.

5. Case Study

The proposed workflow customization technique verifies by developing the user authentication component. This verification examines whether the workflow customization changes to the desired flow. The user authentication component is the component that verifies the information of the user. The information of user is stored in the database or the file of the server.

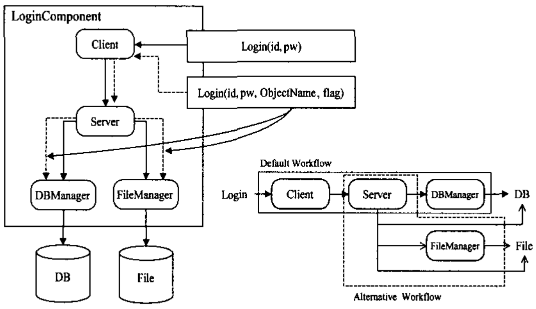


Fig. 17 The User Authentication Component

The Figure 17 shows the permissible workflow when the application calls the interface of component. The default workflows of this component are the Login(id,pw) or the Login(id, pw, objectName, flag). If this default interface of the component is called by the application, the Server object calls the DBManager object that is the object of the default workflow. The customization of the workflow puts the object name and flag of the desired flow to the Login function.

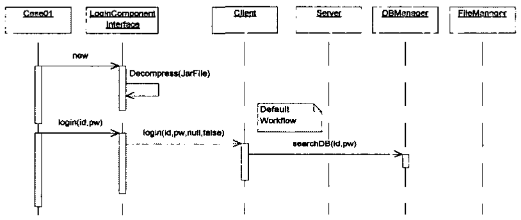


Fig. 18 The Sequence Diagram of the Default Worflow

The Figure 18 is the sequence diagram that represents the default workflow of the user authentication component. If the application Case01 calls the 'Login(id,pw), the interface of the component invokes the 'Login(id,pw,null,False) of the Client. This invocation selects the default workflow that calls the SearchDB(id,pw) of the DBManager object.

The Figure 19 shows the process that is changed the workflow of the component. The application put the changeable object and the flag to the interface Login of component. The flag is set the True. And the inserted object is the object among Server, DBManager, and FileManager object. The component flows into the selected object.

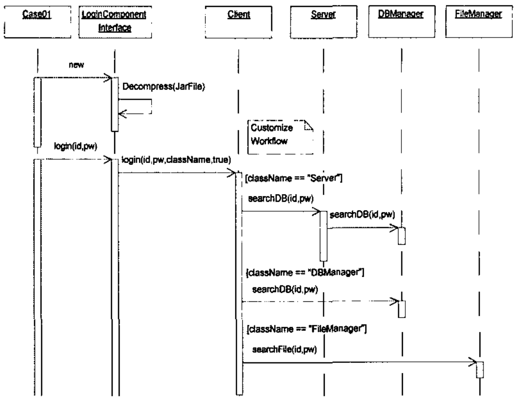
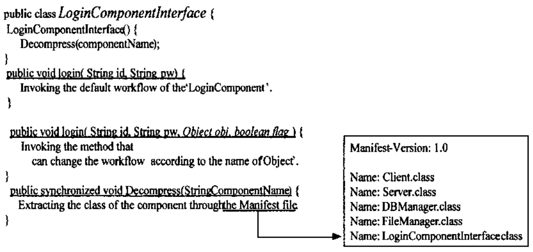


Fig. 19 The Sequence Diagram of the Customization Workflow

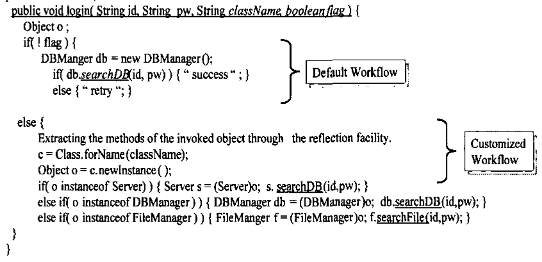


```
public class LoginComponentInterface {
 LoginComponentInterface() {
    Decompress(componentName);
}
 public void login( String id, String pw) {
    Invoking the default workflow of the'LoginComponent'.
 }

 public void login( String id, String pw, Object obj, boolean flag ) {
    Invoking the method that
       can change the workflow  according to the name of'Object'.
}
 public synchronized void Decompress(StringComponentName) {
   Extracting the class of the component throughthe Manifest file
}
```

```
Manifest-Version: 1.0

Name: Client.class
Name: Server.class
Name: DBManager.class
Name: FileManager.class
Name: LoginComponentInterfaceclass
```

Fig. 20 The User Authentication Component

The Figure 20 shows the interface of the user authentication component. This consists of the interface for the default workflow and the interface for the customization.



```
public class client {
 public void login( String id, String  pw, String className, booleanflag ) {
   Object o ;
   if( ! flag ) {
       DBManger db = new DBManager();
          if( db.searchDB(id, pw) ) { " success " ; }
          else { " retry "; }

  else {
      Extracting the methods of the invoked object through   the reflection facility.
      c = Class.forName(className);
      Object o = c.newInstance( );
      if( o instanceof Server) ) { Server s = (Server)o;  s. searchDB(id,pw); }
      else if( o instanceof DBManager ) ) { DBManager db = (DBManager)o;  db.searchDB(id,pw); }
      else if( o instanceof FileManager ) ) { FileManger f = (FileManager)o; f.searchFile(id,pw); }
 }
}
```

Fig. 21 The Client Class

The Figure 21 shows that the workflow is changed according to the passed flag to the client class. When the developer change the workflow of the component, the component creates the new object using the reflection facility.

The created object invokes the method of the object. The invoked method must define the pattern like the get/set method in the business point of view. So the component users can use the common method of the component class.

The Decompress() is the method that extracts the internal classes of the component for the documentation. This method extracts the component using the manifest file of the component and can represent the information of the component using the reflection facility of the Java.

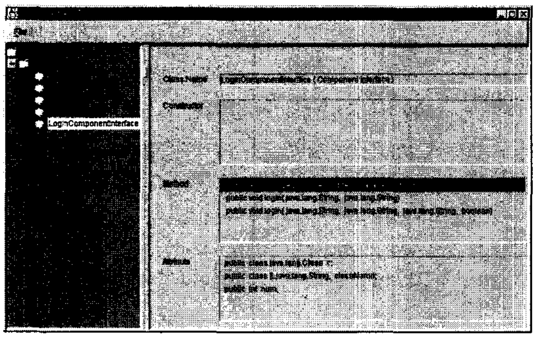
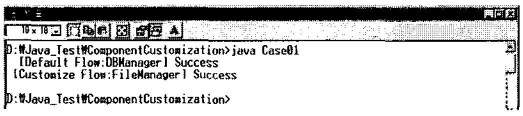


Fig. 22 The Documentation of the User Authen-tication Component

```
public class Case01 {
  public static void main(String[] args) {
      LoginComponentInterface l = new LoginComponentInterface ();
      l.login("cjkim", "pw");
      l.login("cjkim", "pw","FileManager", true);
  }
}
```



```
D:\Java_Test\ComponentCustomization>java Case01
 [Default Flow:DBManager] Success
 [Customize Flow:FileManager] Success

D:\Java_Test\ComponentCustomization>
```

Fig. 23 The Case Study using the Component

Like Figure 22, the tool of the documentation can get the information of class, such as the name of class, constructors, methods, attributes, and so on.

The Figure 23 shows to change the workflow of the component through the component interface. The application uses the component interface to change the workflow. This example shows the default workflow and the changed result of the workflow to the FileManager.

6. Assessments

The Table 1 represents whether the workflow customization of this paper supports in the other component model, such as Java Beans, EJB, and COM. Three component models support the basic facilities completely, such as class packaging, interface, introspection, and encapsulation.

Table 1 The Assessments of the Component Models

Component Models / Factors	JavaBeans	EJB	COM
Class Packaging	O(Jar)	O(EJB-Jar)	O
Interface	O	O	O(IUnknown)
Introspection	O(BeanInfo)	O(Metadata)	O(OLE)
Encapsulation	O	O	O
Objects Plug-In Dynamically	O	O	O
Dynamic Linking	×	×	O
Customize Component	△	△	△
Property Customization	O	O	O
Workflow Customization	×	×	×
Full Support:O , Partial Support:△ , Not Support:×			

Three models can plug the object in the component dynamically. The COM can replace the object within the component dynamically, but the Java Beans and the EJB must be recompiled the replaced component. And three models do not support the mechanism that customizes the workflow of component. Only they support the property customization. So the existing components support partially the customization of the component.

7. Concluding Remarks

Until now, we studied the three techniques that change the workflow of the component in the business point of view. This workflow customization extends the existing customization of the limited facility. *And it is that the component can be reused in* the business point of view. In this paper we proposed the customization technique that selects the workflow and changes the workflow dynamically. These techniques are supported in the black box, so the component can be customized easily by the application.

In the future, we will propose the method that can implement the complex component using the customization technique among components. The complex component must pass the object of any component to other component. So we will study the customization technique that can plug the transmitted *object in other component effectively*

References

[1] Alan W. Brown, Kurt C. Wallnau, The Current State of CBSE, IEEE Software, Sep/Oct 1998.
[2] Robert Orfali, Dan Harkey, Client/Server Programming with Java and CORBA, 2nd ed., John Wiley & Sons, Inc., 1998.
[3] Eduardo Pelegri-Llopart, Laurence P.G. Cable, 'How to be a Good Bean,' Sun Microsystems, Inc., Sep. 1997.
[4] Jun Han, 'Characterization of Components,' International Workshop on Component-Based Software Engineering 1998.
[5] Desmond Francis D'Souza, Alan Cameron Wills, Objects, component, and frameworks with UML : the Catalysis approach, Addison Wesley Longman, Inc., 1999.
[6] Mikio Aoyama, 'New Age of Software Development : How Component-Based Software Engineering Changes the Way of Software Development,' International Workshop on Component-Based Software Engineering 1998.
[7] Digre T., Business Object Component Architecture, *IEEE Software*, pp.60-69, September 1998.
[8] Nicolas, P.R., Component-based Development using CORBA, at URL: http://www.ikonodyne.com/whitecbd/workflow.html, 1998.
[9] Mary C., Kathy W., Alison H., The Java Tutorial Continued, Addion-Wesley, 1999.
[10] Klaus Bergner, Andreas Rausch, Marc Sihling, 'Componentware The Big Picture,' International Workshop on Component-Based Software Engineering 1998.
[11] Brown A. W. and Wallnau K. C., The Current State of CBSE, *IEEE Software,* pp.37-46, Sept./Oct. 1998.
[12] Short K., *Component Based Development and Object Modeling,* Sterling Software, 1997.

김 철 진
1996년 경기대학교 전산학 학사. 1998년 숭실대학교 전산학 석사. 1998년 ~ 현재 숭실대학교 대학원 전자계산학과 박사과정. 관심분야는 객체지향 개발방법론(OMT, UML). 웹 기반 분산 객체 컴퓨팅(CORBA, Client-Server Web)

김 수 동
정보과학회논문지 : 소프트웨어 및 응용 제 27 권 제 3 호 참조